# SOFTWARE CONTROLLED PRE-EXECUTION IN A MULTITHREADED PROCESSOR

## CROSS-REFERENCE TO RELATED APPLICATIONS

**[0001]** This application relates to non-provisional U.S. application Serial No. 09/584,034, filed on May 30, 2000, and entitled "Slack Fetch to Improve Performance in a Simultaneous and Redundantly Threaded Processor" and to provisional application Serial No. 60/198,530, filed on April 19, 2000, and entitled "Transient Fault Detection Via Simultaneous Multithreading," the teachings of both of which are incorporated herein by reference.

## STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH OR DEVELOPMENT

**[0002]** Not applicable.

## BACKGROUND OF THE INVENTION

### Field of the Invention

**[0003]** The present invention generally relates to microprocessors. More particularly, the present invention relates to a computer capable of running multiple threads that can concurrently process instructions in at least two separate threads. More particularly still, the invention relates to pre-executing at least a portion of a program as separate threads to avoid cache misses.

### Background of the Invention

**[0004]** Most modern computer systems include at least one central processing unit ("CPU") and a main memory. Multiprocessor systems include more than one processor and each processor typically has its own memory which may or may not be shared by other processors. The speed at

which the CPU can decode and execute instructions and operands depends upon the rate at which the instructions and operands can be transferred from main memory to the CPU. In an attempt to reduce the time required for the CPU to obtain instructions and operands from main memory, many computer systems include a cache memory coupled between the CPU and main memory.

**[0005]** A cache memory is a relatively small, high-speed memory (compared to main memory) buffer that is used to temporarily hold those portions of the contents of main memory which it is believed will be used in the near future by the CPU. The main purpose of a cache is to shorten the time necessary to perform memory accesses, both for data and instructions. Cache memory typically has access times that are several or many times faster than a system's main memory. The use of cache memory can significantly improve system performance by reducing data access time, therefore permitting the CPU to spend far less time waiting for instructions and operands to be fetched and/or stored.

**[0006]** A cache memory, typically comprising some form of random access memory ("RAM") includes many blocks (also called lines) of one or more words of data. Associated with each cache block in the cache is a tag. The tag provides information for mapping the cache line data to its main memory address. Each time the processor makes a memory reference (*i.e.*, load or store), a tag value from the memory address is compared to the tags in the cache to see if a copy of the requested data resides in the cache. If the desired memory block resides in the cache, then the cache's copy of the data is used in the memory transaction, instead of the main memory's copy of the same data block. However, if the desired data block is not in the cache, the block must be retrieved from the main memory and supplied to the processor. A copy of the data also is stored in the cache.

[0007] Because the time required to retrieve data from main memory is substantially longer than the time required to retrieve data from cache memory, it is highly desirable have a high cache "hit" rate. Although cache subsystems advantageously increase the performance of a processor, not all memory references result in a cache hit. A cache "miss" occurs when the targeted memory data has not been cached and must be retrieved from main memory. Thus, cache misses detrimentally impact the performance of the processor, while cache hits increase the performance. Anything that can be done to avoid cache misses is highly desirable, and there still remains room for improvement in this regard.

## BRIEF SUMMARY OF THE INVENTION

[0008] The problems noted above are solved in large part using a processor (*e.g.*, a simultaneous multithreading processor) that can run a program in one thread (called the "main" thread) and at least a portion of the same program in another thread (called the "pre-execution" thread). In general, the concepts apply to any computer that can execute multiple threads of control that share common cache structures. The program in the main thread includes instructions that cause the processor to start and stop pre-execution threads and direct the processor as to which part of the program is to be run through the pre-execution threads. Preferably, such instructions cause the pre-execution thread to run ahead of the main thread in program order (*i.e.*, fetching an instruction from the program before the main thread fetches the same instruction). In this way, preferably any cache miss conditions that are encountered by the pre-execution thread are resolved before the main thread requires that same data. Therefore, the main thread should encounter few or no cache miss conditions, and overall performance is increased.

[0009] In accordance with one embodiment, the invention is embodied as a computer system which includes a simultaneous multithreading processor, an I/O controller coupled to said

processor, an I/O device coupled to said I/O controller, and a main system memory coupled to said processor, wherein said processor processes a program in a main thread that includes instructions which cause the processor to spawn a pre-execution thread in which at least a portion of the same program executes, said pre-execution thread runs concurrently with the main thread, but ahead of the main thread in program order. Further, another embodiment of the invention includes a method of running a program in a simultaneous multithreading processor comprising inserting pre-execution thread instructions in the program, spawning a pre-execution thread when designated by the inserted instructions, and running said pre-execution thread concurrently with a main thread wherein both the pre-execution and the main threads including instructions from the same program, the pre-execution thread running ahead of the main thread.

**[0010]** In addition to reducing the potential for cache misses to occur, the technique described herein can be used to mitigate the effects of branch mispredictions. These and other benefits will become apparent upon reviewing the following disclosure.

## BRIEF DESCRIPTION OF THE DRAWINGS

**[0011]** For a detailed description of the preferred embodiments of the invention, reference will now be made to the accompanying drawings in which:

**[0012]** Figure 1 is a diagram of a computer system constructed in accordance with the preferred embodiment of the invention and including a simultaneous multithreading processor;

**[0013]** Figure 2 is a block diagram of the simultaneous multithreading processor from Figure 1 in accordance with the preferred embodiment;

**[0014]** Figure 3 illustrates how a program can be modified to reduce the performance impact of cache misses; and

[0015] Figure 4 illustrates a preferred embodiment of embedding software instructions in a program to spawn pre-execution threads to reduce the potential for cache misses in the main program.

## NOTATION AND NOMENCLATURE

[0016] Certain terms are used throughout the following description and claims to refer to particular system components. As one skilled in the art will appreciate, microprocessor companies may refer to a component by different names. This document does not intend to distinguish between components that differ in name but not function. In the following discussion and in the claims, the terms "including" and "comprising" are used in an open-ended fashion, and thus should be interpreted to mean "including, but not limited to...". Also, the term "couple" or "couples" is intended to mean either an indirect or direct electrical connection. Thus, if a first device couples to a second device, that connection may be through a direct electrical connection, or through an indirect electrical connection via other devices and connections.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

[0017] In accordance with a preferred embodiment of the invention, a simultaneous multithreading processor is used to reduce the potential for cache misses to occur. Figures 1 and 2 show a computer system that incorporates such a processor and Figures 3 and 4 illustrate its use in solving the problems noted above. In general, the principles discussed below can be applied to any type of computer that executes multiple threads of control that share some common cache structures. For sake of explanation, a simultaneous multithreading processor is used to explain the principles.

[0018] Referring now to Figure 1, a computer system 90 is shown including a pipelined, simultaneous multithreading threaded ("SMT") processor 100 constructed in accordance with the preferred embodiment of the invention. Besides processor 100, computer system 90 may also include dynamic random access memory ("DRAM") 92, an input/output ("I/O") controller 93, and various I/O devices which may include a floppy drive 94, a hard drive 95, a keyboard 96, and the like. The I/O controller 93 provides an interface between processor 100 and the various I/O devices 94-96. The DRAM 92 can be any suitable type of memory devices such as RAMBUS™ memory. In addition, SMT processor 100 may also be coupled to one or more other processors if desired.

[0019] Figure 2 shows the SMT processor 100 of Figure 1 in greater detail. The embodiment shown in Figure 2 is exemplary only—any SMT processor architecture consistent with the principles discussed herein is acceptable. Referring to Figure 2, processor 100 preferably comprises a pipelined architecture which includes a series of functional units, arranged so that several units can be simultaneously processing appropriate parts of multiple instructions. As shown, the exemplary embodiment of processor 100 includes a fetch unit 102, one or more program counters 106, an instruction cache 110, decode logic 114, register rename logic 118, floating point and integer registers 122, 126, a register update unit 130, execution units 134, 138, and 142, and a data cache 146.

[0020] Fetch unit 102 uses one or more program counters 106 for assistance as to which instructions to fetch. Being a multithreading processor, the fetch unit 102 preferably can simultaneously fetch instructions from multiple threads. A separate program counter 106 is associated with each thread. Each program counter 106 is a register that contains the address of the next instruction to be fetched from the corresponding thread by the fetch unit 102. Figure 2 shows

two program counters 106 to permit the simultaneous fetching of instructions from two threads. It should be recognized, however, that additional program counters can be provided to fetch instructions from more than two threads simultaneously.

**[0021]** As shown, fetch unit 102 includes branch prediction logic 103. The branch prediction logic 103 permits the fetch unit 102 to speculate ahead on branch instructions. In order to keep the pipeline full (which is desirable for efficient operation), the branch predictor logic 103 speculates the outcome of a branch instruction before the branch instruction is actually executed. Branch predictor 103 generally bases its speculation on previous instructions. Any suitable speculation algorithm can be used in branch predictor 103.

**[0022]** Referring still to Figure 2, instruction cache 110 provides a temporary storage buffer for the instructions to be executed. Decode logic 114 retrieves the instructions from instruction cache 110 and determines the type of each instruction (*e.g.*, add, subtract, load, store, etc.). Decoded instructions are then passed to the register rename logic 118 which maps logical registers onto a pool of physical registers.

**[0023]** The register update unit ("RUU") 130 provides an instruction queue for the instructions to be executed. The RUU 130 serves as a combination of global reservation station pool, rename register file, and reorder buffer. The RUU 130 breaks load and store instructions into an address portion and a memory (*i.e.*, register) reference. The address portion is placed in the RUU 130, while the memory reference portion is placed into a load/store queue (not specifically shown in Figure 2).

**[0024]** The floating point register 122 and integer register 126 are used for the execution of instructions that require the use of such registers as is known by those of ordinary skill in the art.

These registers 122, 126 can be loaded with data from the data cache 146. The registers also provide their contents to the RUU 130.

**[0025]** As shown, the execution units 134, 138, and 142 comprise a floating point execution unit 134, a load/store execution unit 138, and an integer execution unit 142. Each execution unit performs the operation specified by the corresponding instruction type. Accordingly, the floating point execution units 134 execute floating instructions such as multiply and divide instruction while the integer execution units 142 execute integer-based instructions. The load/store units 138 perform load operations in which data from memory is loaded into a register 122 or 126. The load/ store units 138 also perform store operations in which data from registers 122, 126 is written to data cache 146 and/or DRAM memory 92 (Figure 1).

**[0026]** The architecture and components described herein are typical of microprocessors, and particularly pipelined, multithreaded processors. Numerous modifications can be made from that shown in Figure 2. For example, the locations of the RUU 130 and registers 122, 126 can be reversed, if desired.

**[0027]** In general, any memory reference (*e.g.*, a store or load) in a program will result in a cache miss if the needed data is not currently present in data cache 146. However, in accordance with the preferred embodiment of the invention, the SMT processor 100 can be used to spawn (*i.e.*, create, initialize and execute) a thread which resolves cache misses ahead of the main program. This spawned thread preferably runs concurrently with, but ahead of, the thread in which the main program executes. When the main program then encounters a memory reference, the needed data has already been cached by the previously spawned thread. In this context, the thread in which the main program executes is called the "main" thread and the thread which is spawned to resolve cache misses is called the "pre-execution" thread.

[0028] This technique can be understood with reference to Figure 3. Figure 3 shows the same portion of a program in two different forms 202 and 220. Each program form 202, 220 includes two program segments 204 and 206. Program 202 represents the program without the ability to spawn pre-execution threads. Program 220 represents program 202 modified to spawn pre-execution threads at desired points during program execution. As shown, the modifications include start and stop pre-execution thread instructions 222 and 226, as well as a label 224. In accordance with the preferred embodiment, the instruction to start a pre-execution thread is of the form "PreExecute_Start(label)" where "label" is a value that informs the program counter 106 where to begin executing the pre-execution thread. In the example of Figure 3, "label" is "LIST2" which is identified by numeral 224. Thus, at run-time, when PreExecute_Start(LIST2) is executed by the main thread, a pre-execution thread will be spawned to execute the program starting at the label LIST2. The pre-execution thread then continues executing code segment 206 from LIST2 until the PreExecute_Stop() instruction 226 is encountered.

[0029] While the main thread is executing code segment 224, the pre-execution thread runs ahead of the main thread and executes code segment 206. Cache misses encountered during the execution of segment 206 by the pre-execution thread preferably are resolved before the main thread encounters the same memory references when it executes code segment 206. Broadly speaking, the pre-execution thread resolves one or more memory references and causes the requested data to be in data cache 146 before the main thread needs the data.

[0030] The start/stop instructions shown in Figure 3 represent one way to cause a thread to be spawned. In general, any mechanism to spawn a pre-execution thread is within the scope of this disclosure. For example, a pre-execution thread could simply be programmed to spin on a synchronization variable. Then, the main thread can reset the synchronization variable when there

are instructions to pre-execution thereby causing the pre-execution to pre-execute such instructions.

**[0031]** The process of spawning a pre-execution thread generally requires the pre-execution thread to have access to the register data from the main thread. Accordingly, the main thread can write values of the registers that will be needed by the pre-execution thread into memory. Then, the pre-execution thread will load these values from memory before starting pre-execution.

**[0032]** Besides using a stop instruction, such as PreExecute_Stop() 226, to terminate a pre-execution thread, there are numerous other ways to terminate a pre-execution thread. For example, a PreExecute_Cancel(T) instruction can be used in the main thread. When the main thread encounters this instruction, the pre-execution thread will be terminated. Additionally, when the pre-execution thread tries to pre-execute a program counter (PC) that is out of the acceptable range of instructions as imposed by the operating system, the pre-execution thread will be terminated. Also, the pre-execution thread can be terminated when the hardware detects the main thread has caught up to the pre-execution thread. Finally, but without limitation, the pre-execution thread can be terminated when the number of instructions it has run exceeds a limit. This makes sure that all pre-execution threads eventually terminate.

**[0033]** Figure 4 illustrates a method 250 of implementing pre-execution threads. The method 250 includes steps 252, 254, 256 and 258. In step 252, the pre-execution instructions are inserted into the program. These instructions include both the start and stop instructions, as well as the labels. These instructions can be inserted by the programmer or by a compiler. Then, in step 254, the processor 100 spawns pre-execution thread(s) when and where indicated by the pre-execution start instructions. The start pre-execution thread instruction preferably returns the identity of the spawned thread or a pre-designated value (*e.g.*, -1) if there is are insufficient hardware resources

available to spawn another thread. To spawn a pre-execution thread, various registers such a thread will use preferably are initialized. The registers should be initialized in accordance with the registers used by the main thread so that the pre-execution thread executes identically to the main thread. To this end, the preferred implementation of the invention uses software to copy the register values from the main thread to the pre-execution thread. More specifically, a software routine is constructed to save the register values of the main thread to memory and then retrieve them back from the memory to the registers of the pre-execution thread.

**[0034]** Step 256 acknowledges that the pre-execution thread(s) run concurrently with the main thread. While executing, it is desirable for the pre-execution thread to avoid speculative pre-execution from corrupting the program correctness. Accordingly, the preferred implementation ignores all exceptions such as invalid load addresses, division by zero, etc. generated during pre-execution. In this way, the main thread will not encounter any additional exceptions. In addition, pre-execution threads should not be permitted to modify memory contents (*e.g.*, stores). Thus, in accordance with the preferred embodiment, all store instructions encountered during pre-execution are dropped by processor 100 after they are decoded. That is, they are executed as no-ops ("NOPs").

**[0035]** Instead of simply dropping pre-execution stores, a "scratchpad" buffer can be added to the hardware to buffer the effect of stores during pre-execution. Instead of writing into the cache and the memory, stores in a pre-execution thread will write into the scratchpad buffer which may be coupled to the load/store units 138. In this way, loads in the same pre-execution thread will look up data in both the cache and scratchpad. The main advantage of this approach is that pre-executed loads can observe the effect of earlier stores from the same thread, which may be important to generating future data addresses or maintaining the correct control flow. For instance,

if a procedure with a pointer-type argument (*e.g.*, the head of a linked list) is called during pre-execution, both the pointer value and the return address could be passed through the stack. Thus, it would be desirable to be able to read both values back from the stack in the procedure during pre-execution so that the correct data item can be fetched and the procedure can eventually return to its caller as well.

**[0036]** It should be apparent that, although the threads run concurrently with one another, the pre-execution thread preferably does not run the same instructions as the main thread at the same time. That is, by causing the pre-execution thread to jump ahead of the main thread and begin executing at a later point in program order, a "slack" between the two threads is automatically created. This slack facilitates the pre-execution thread to resolve cache misses before the main program requests the data. Finally, in step 258 the pre-execution thread ceases when it encounters its stop execution instruction.

**[0037]** The pre-execution capability described herein is software implemented and controlled as should be apparent. Common uses of software-controlled pre-execution include pre-execution multiple pointer chains, pre-executing loops involving non-affined array references, pre-execution multiple procedure calls, and pre-executing multiple control-flow paths, to name a few. Further, pre-execution may also be useful to avoid branch mispredictions, which is a problem well known to those of ordinary skill in the art and described in U.S. Patent Application serial no. 09/584,034 entitled "Slack Fetch to Improve Performance in a Simultaneous and Redundantly Thread Processor."

**[0038]** Accordingly, the preferred embodiment of the invention provides a significant performance increase in a processor. The above discussion is meant to be illustrative of the principles and various embodiments of the present invention. Numerous variations and

modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.